# Supplemental Material: EasyPDP: An Efficient Parallel Dynamic Programming Runtime System for Computational Biology

Shanjiang Tang, Ce Yu*, Jizhou Sun, Bu-Sung Lee, Tao Zhang, Zhen Xu, Huabei Wu

**Abstract**—Dynamic programming is a popular and efficient technique in many scientific applications such as computational biology. Nevertheless, its performance is limited due to the burgeoning volume of scientific data, and parallelism is necessary and crucial to keep the computation time at acceptable levels. The intrinsically strong data dependency of dynamic programming makes it difficult and error-prone for the programmer to write a correct and efficient parallel program. Therefore this paper builds a runtime system named EasyPDP aiming at parallelizing dynamic programming algorithms on multi-core and multi-processor platforms. Under the concept of software reusability and complexity reduction of parallel programming, a DAG Data Driven Model is proposed, which supports those applications with a strong data interdependence relationship. Based on the model, EasyPDP runtime system is designed and implemented. It automatically handles thread creation, dynamic data task allocation and scheduling, data partitioning, and fault tolerance. Five frequently used DAG patterns from biological dynamic programming algorithms have been put into the DAG pattern library of EasyPDP, so that the programmer can choose to use any of them according to his/her specific application. Besides, an ideal computing distribution model is proposed to discuss the optimal values for the performance tuning arguments of EasyPDP. We evaluate the performance potential and fault tolerance feature of EasyPDP in multi-core system. We also compare EasyPDP with other methods such as Block-Cycle Wavefront(BCW). The experimental results illustrate that EasyPDP system is fine and provides an efficient infrastructure for dynamic programming algorithms.

**Index Terms**—Dynamic Programming, EasyPDP, DAG Data Driven Model, fault tolerance, DAG pattern, multi-core, block-cycle.

## APPENDIX A
## THE DP ALGORITHM AND CLASSIFICATION

DP is an important algorithm design technique in computational biology. It solves the problem by decomposing the problem into a set of interdependent subproblems, and using their results to solve larger subproblems until the entire problem is solved. In general, the solution to a DP problem is expressed as a minimum(or maximum) of possible alternative solutions. Each of these alternative solutions is constructed by composing one or more solutions to subproblems. If $r$ represents the cost of a solution composed of subproblems $x_1$,$x_2$,...,$x_l$, then $r$ can be written as $r=g(f(x_1),f(x_2),...,f(x_l))$, where the function $g$ is called the composition function, and its nature depends on the problem. If the optimal solution to each problem is determined by composing optimal solutions to the subproblems and selecting the minimum(or maximum), the formulation is said to be a DP formulation[10].

Based on the dependencies between subproblems in a DP formulation, there are various classification criteria. Grama et al.[10] presents a classification of DP: DP is considered as a multistage problem composed of many subproblems. If the subproblems located on all levels depend only on the results from the immediately preceding levels, it is called serial; otherwise, it is called nonserial. There is recursive equation called a functional equation, which represents the solution to an optimization problem. If a functional equation contains a single recursive term, the DP is monadic; otherwise, if it contains multiple recursive terms, it is polyadic. Based on this classification criterion, four classes of DP are defined: serial monadic, serial polyadic, nonserial monadic, and nonserial polyadic. Considering the cache-efficient parallel execution, Chowdhury et al.[11] provide cache-efficient algorithms for three different classes of DP: LDDP (Local Dependency DP) problem, GEP (Gaussian Elimination Paradigm), and Parenthesis problem, each of which embraces one class of DP applications[12]. We consider another classification method. That is, the DP algorithms can be classified in terms of the matrix dimension and the dependency relationship of each cell on the matrix[30]: A DP algorithm is called a *tD/eD* algorithm if its matrix dimension is $t$ and each matrix cell depends on $O(n^e)$ other cells. If a DP algorithm is a

- *S.J. Tang, C. Yu, J.Z. Sun, T. Zhang, Z. Xu, H.B. Wu are with the School of Computer Science&Technology, Tianjin University, Tianjin 300072, China.*
  *E-mail: {tashj, yuce, jzsun}@tju.edu.cn.*
- *B.S. Lee is with the School of Computer Engineering, Nanyang Technological University, Singapore.*
  *E-mail: ebslee@ntu.edu.sg.*
- *C. Yu* (yuce@tju.edu.cn): the correspondence author.*

TABLE 1: Some Popular DP Algorithms for Computational Biology

| Algorithm | Application | Reference |
|---|---|---|
| Smith-Waterman algorithm with linear and affine gap penalty | Genome local alignment | [19] |
| Syntenic alignment | Generalized genome global alignment | [18] |
| Smith-Waterman algorithm with general gap penalty | Genome local alignment | [19][20] |
| Nussinov algorithm | RNA base pair maximization | [21] |
| Viterbi Algorithm | Gene sequence alignment using HMMs, Multiple sequence alignment | [22] |
| Double DP algorithm | Protein threading | [23] |
| Spliced Alignment | Gene finding | [24] |
| Zuker Algorithm | RNA secondary structure prediction | [25] |
| CYK Algorithm | RNA secondary structure alignment | [20] |

*tD/eD* algorithm, it takes time $O(n^{t+e})$ provided that the computation of each term takes a constant duration of time. For example, three DP algorithms are defined as follows:

**Algorithm 1.** (2D/0D): Given F[*i*,0] and F[0,*j*] for

$$0 \leq i,j \leq n,$$

$$F[i,j] = min\{F[i-1,j]+x_i, F[i,j-1]+y_j, F[i-1,j-1]+z_{ij}\},$$

where $x_i$,$y_j$ and $z_{ij}$ are computed in constant time.

**Algorithm 2.** (2D/1D): Given *c(i,j)* for $1 \leq i < j \leq n$, F[*i*,*i*]

$$=0 \text{ for } 1 \leq i \leq n,$$

$$F[i,j] = c(i,j) + min\{F[i,k-1] + F[k,j]\},$$

where $1 < k \leq j$ and *c(i,j)* is computed in constant time.

**Algorithm 3.** (2D/2D): Given *c(i,j)* for $1 \leq i < j \leq 2n$, F[*i*,0]

and F[0,*j*] for $1 \leq i,j \leq n$,

$$F[i,j] = min\{F[i^{'},j^{'}] + c(i^{'}+j^{'}, i+j)\},$$

where $0 \leq i^{'} < i$, $0 \leq j^{'} < j$ and *c(i,j)* is computed in constant time.

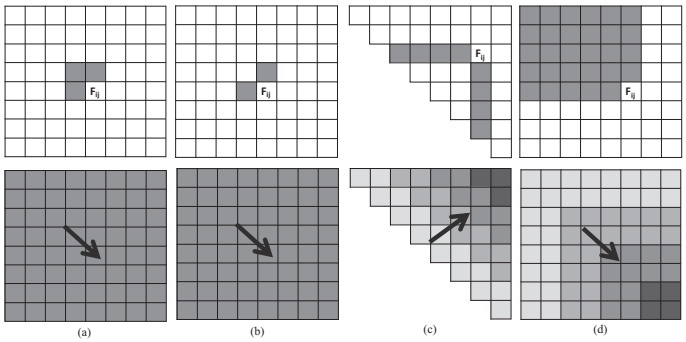



Fig. 1: The computation dependency relationship and distribution of load computation density along computation shift direction. (a) and (b) are *2D/0D* DP algorithms, while (c) is *2D/1D* DP algorithm and (d) is *2D/2D* DP algorithm.

For a DP algorithm, if each matrix cell is computed from the same number of other matrix cells, then the DP is a regular one, otherwise, we call it irregular one.

There are many DP algorithms in computational biology. Some popular DP algorithms for computational biology are shown in Table 1. DP is applied for sequence comparison with numerous variations, determining the intron/exon structure of genes and assembling DNA sequences from overlapping fragments. As shown in Figure 1, it is the computation dependency relationship and distribution of load computation density along computation shift direction for some popular algorithms. By using increasingly blacking shades to indicate computational load density changes, we could notice that the *2D/0D* DP algorithms are regular ones, while the *2D/iD* ($i \geq 1$) DP algorithms are irregular. We concentrate on the parallelization of DP algorithms of the type *2D/iD* ($i \geq 0$), which are important DP algorithms for many applications.

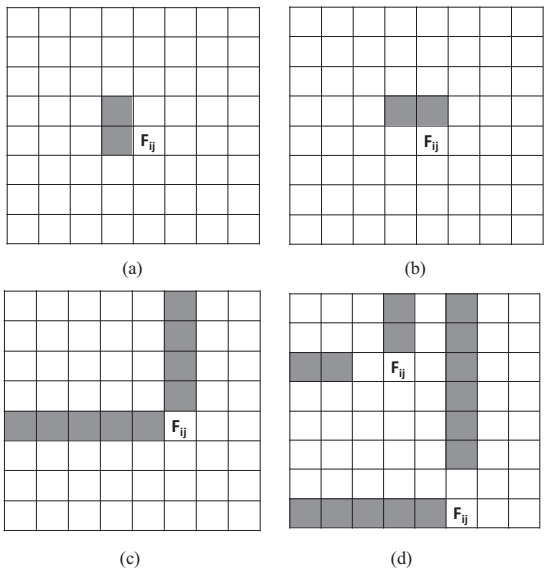



Fig. 2: The computation dependency relationship for some DP algorithms.

## APPENDIX B
## EXAMPLE AND REUSABILITY

Figure 3 presents a sample source code for parallelism of smith-waterman algorithm with linear and affine gap penalty[19] with EasyPDP framework. User only needs to do some initialization work and puts his/her attention on a specific application, while the parallel part has been implemented by EasyPDP. Specifically, user first implements his/her DP application in an application-specific function(defined by the user)(lines 2-37). Then user selects a DAG Pattern from the DAG pattern library(lines 43-45) accordingly and does some required argument settings (lines 46-57). After configurations, s/he can call EasyPDP scheduler function to do parallel processing(lines 58-59). All parallel details such as dynamic scheduling and distribution mechanisms, which are application-independent and reusable, are transparent to the user. From the user's perspective, the parallel programming based on EasyPDP is very similar to 'serial' programming. Moreover, if user wants now to implement affine gap penalties (Gotoh82 algorithm), s/he would only have to change the application-specific function(`smith_waterman_alignment` in Figure 3).

```
1: #include "EasyPDP_Scheduler.h"
2: /**
3:  * The application-specific function.
4:  * @param arg   Pointer to data_blocks arg
5:  */
6: void smith_waterman_alignment(void* arg) {

7:  data_blocks* data_block = (data_blocks*)arg;
8:  int data_row = data_block->data_row;L
9:  int data_col = data_block->data_col;
10: int block_row = data_block->block_row;
11: int block_col = data_block->block_col;
12: int x = data_block->pos.x;
13: int y = data_block->pos.y;
14: int w;
15: for(int i=x; i<data_row&&i<x+block_row; i++){
16:   for(int j=y; j<data_col&&j<y+block_col; j++){
17:     M[i][j] = 0;
18:     if(i==0 || j==0)
19:       M[i][j] = 0;
20:     if(i >0 && j > 0){
21:       w = smith_waterman_getScore(seq1[i],seq2[j]);
22:       if(M[i][j] < M[i-1][j-1] + w)
23:         M[i][j] = M[i-1][j-1] + w;
24:     }
25:     if(i > 0){
26:       w = smith_waterman_getScore(seq1[i],'-');
27:       if(M[i][j] < M[i-1][j] + w)
28:          M[i][j] = M[i-1][j] + w;
29:     }
30:     if(j > 0){
31:       w = smith_waterman_getScore('-',seq2[j]);
32:       if(M[i][j] < M[i][j-1] + w)
33:         M[i][j] = M[i][j-1] + w;
34:     }
35:   }
36: }
37:}

38:int main(int argc, char** argv) {
39: scheduler_args_t sched_args;
40: DAGPattern_args_t DAGPattern_arg;

41: //The initialization function.
42: smith_waterman_init();

43: //Select DAG Pattern.
44: DAGPattern_arg.DAG_pattern_id = Left_Upper_DAG;
45: sched_args.DAGPattern_arg = &DAGPattern_arg;

46: //Set thread_num to be the number of processors.
47: sched_args.thread_num= getNumberOfCpus();

48: sched_args.dp_data   = (void*)M;
49: sched_args.data_col  = sequence1.length + 1;
50: sched_args.data_row  = sequence2.length + 1;
51: sched_args.block_col = sequence1.length > 500?
     (sequence1.length+1)/10 : sequence1.length + 1;
52: sched_args.block_row = sequence2.length > 500?
     (sequence2.length+1)/10 : sequence2.length + 1;
53: sched_args.timeout   = 30;

54: //Set DP application-specific function.
55: sched_args.process=(void*)smith_waterman_alignment;

56: //Use default mapping function.
57: sched_args.DAG_pattern_node_data_mapping = NULL;

58: //Call EasyPDP scheduler for parallel processing.
59: EasyPDP_scheduler(&sched_args);

60: //Trace back results
61: smith_waterman_traceback();

62: return 0;
63:}
```

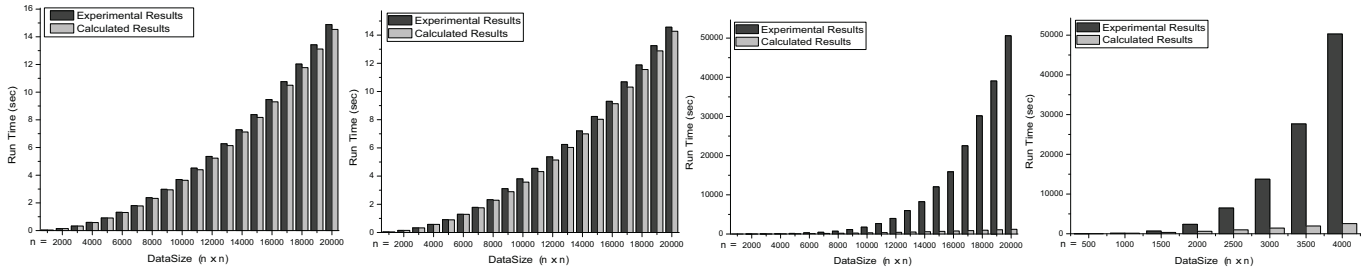
Fig. 3: The sample source code for smith-waterman algorithm with linear and affine gap penalty.

## APPENDIX C
## PERFORMANCE EVALUATION

This section presents the performance evaluation results for EasyPDP running on Dell PowerEdge 2950 Dual Quad Core server with Xeon E5310 processors of 64K L1 cache and 4096K L2 cache. Four popular DP algorithms taken from Table 1 are evaluated. Specifically, the Smith-Waterman algorithm with linear and affine gap penalty (SWLAG), and Syntenic alignment(SA) algorithm are regular DP algorithms, whereas the Smith-Waterman algorithm with general gap penalty (SWGG), and Viterbi Algorithm(VA) are irregular DP algorithms. The computation dependency relationships for SWLAG, SA, VA, SWGG correspond to (a),(b),(d) of Figure 1, and (c) of Figure 4 respectively. The vtune[32] profiler is used for sampling the number of cache miss events occurred in EasyPDP.

### C.1 Dependency on Data Size

Figure 4 presents the experimental and calculated run time results for EasyPDP as we increase the input data size when the block size and the number of worker threads are unchanged. Particularly, the calculated result refers to the value of Formula (3) for the regular DP alogrithm in Section 4.2.5. Moreover, the formula is initialized based on the two random groups of experimental results. There is a linear relationship between the calculated run time and data size for the regular DP algorithm according to the Formula (3). For (a) and (b) of Figure 4, it is obvious that the experimental result is much closer to the calculated result from the Formula (3). In contrast, (c) and (d) of Figure 4 show

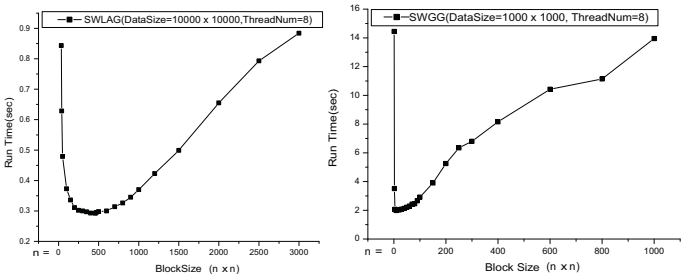



(a) SWLAG(BlockSize=50 × 50,ThreadNum=8) (b) SA(BlockSize=50 × 50,ThreadNum=8) (c) SWGG(BlockSize=50 × 50,ThreadNum=8) (d) VA(BlockSize=50 × 50,ThreadNum=8)

Fig. 4: The dependency on data size. Both the experimental and calculated run time results with different data sizes are given for four DP algorithms. Specifically, Figure (a) and (b) are results for the regular DP algorithms, (c) and (d) are results for the irregular DP algorithms.

that the experimental run time results for the irregular DP algorithms are much large and increase greatly with increasing time intervals compared to the regular DP algorithms. Furthermore, the compared results illustrated that the Formula (3) is no longer suitable for the irregular DP algorithm. The reason is that, for the irregular DP algorithm, the workload for each matrix cell is undetermined and often increases greatly along the anti-diagonal, and thereby the total computational volume increases sharply as we enlarge the input data size.

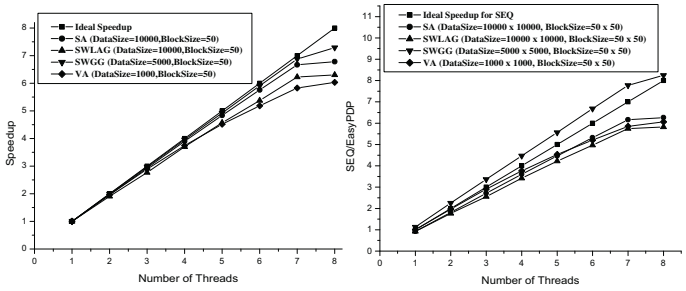
### C.2 Dependency on Block Size




(a) (b)

Fig. 5: The EasyPDP run time results with different block sizes. In Figure (a), the most suitable block size is between 400 × 400 and 500 × 500. While in Figure (b), the most suitable block size is between 8 × 8 and 20 × 20.

The *BlockSize* is a critical performance argument in DP algorithm parallelization. Its setting is a tradeoff between the load balancing and communication time. Both too big and too small values of *BlockSize* will adversely affect the program performance. And often the value of the most suitable block size for regular DP algorithm is much bigger than that of the irregular DP algorithm, due to workload of the data blocks. In general, the practical computation workload for a irregular data block is often many times or more than that for a regular data block with the same block size. The running time results for




(a) (b)

Fig. 6: The dependency on the number of threads. (a) The speedups and comparisons against EasyPDP when 'the number of worker threads' is set to be one for four DP algorithms as we scale the number of worker threads. (b) The comparisons against the sequential iterative code as we scale the number of EasyPDP worker threads for four DP algorithms.

various block sizes are illustrated in Figure 5. We can observe that each of them has a most suitable block size, and for regular SWLAG DP algorithm, its most suitable block size is between 400 × 400 and 500 × 500, whereas the most suitable block size for irregular SWGG DP algorithm is between 8 × 8 and 20 × 20.

### C.3 Dependency on Number of Threads

Figure 6(a) presents the speedups and comparisons against EasyPDP when 'the number of worker threads' is set to be one as we scale the number of worker threads for four popular DP algorithms in the dual quad cores system. It is obvious that all the speedup curves are much close to the ideal speedup curve except their last points for which the number of worker threads is 8. The phenomenon illustrates that the EasyPDP has a good scalability in its performance improvement. We know that when the number of threads is equal to the number of system cores, the speedup is often the best. Since our EasyPDP is implemented as the master-slave model, the number of application threads in fact should be 9 when we set the number of worker threads to be 8, which just

exceeds the number of processor cores by one.

Figure 6(b) gives out the comparisons between the sequential iterative code and EasyPDP when we scale the number of EasyPDP worker threads. We can note that the curve of SWGG is above the ideal speedup line, while the others are not. The reason is that the affection of cache miss for the algorithm SWGG in EasyPDP is non-negligible, while other algorithms are not(See detail discussions about cache miss in APPENDIX C.4). By comparing curves between (a) and (b) of Figure 6 for algorithms SWLAG, SA, and VA, it is apparent that the curves in (b) of Figure 6 are a bit further away from the ideal speedup curve. This is due to the influence from the EasyPDP overhead (See the discussion about EasyPDP overhead in APPENDIX C.4).

### C.4 EasyPDP Overhead and Cache Miss

If we view the sequential iterative code(SEQ for short) as a non-overhead implementation for DP algorithms, we could obtain the overhead of EasyPDP in multi-core systems by comparing the run time against SEQ with the number of worker threads to be one and the value of block size to be that of input data size. As a generic framework system, the overhead of EasyPDP mainly attributes to those factors such as DAG operations, fault tolerance mechanism, worker pool management, etc. As shown in Figure 7, it is apparent that the overhead of EasyPDP is minor(about $1\% \sim 4\%$) in contrast to SEQ for four DP algorithms in multi-core systems.

As discussed in [11][12][13][14], cache-efficiency is very crucial for DP algorithms running in CMP systems[47]. The application of completely iterative DP often results in inefficient cache usage. In [11][12], the authors proposed a cache-efficient divide-and-conquer algorithm which divides the DP problem into lots of subproblems and solves them concurrently in one direction. In contrast, our EasyPDP automatically partitions the DP into lots of blocks according to the argument *BlockSize* set by the user, and solves each by working threads iteratively. Therefore, *BlockSize* is a key argument affecting the number of cache misses for EasyPDP. Here, we design experiments by changing the value of *BlockSize* while setting the number of threads to one for four DP algorithms. The results are shown in Figure 8.

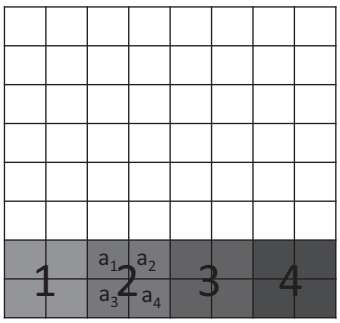



Fig. 9: Cache miss analysis for four DP algorithms.

Indeed, the argument *BlockSize* has a few affection to the reduction of the number of cache misses for SWLAG, SA and VA algorithms. However, it dramatically affects the number of cache misses to the irregular SWGG algorithm. We take Figure 9 as an example to analyze the cache miss. The computation dependency relationships for SWLAG, SA, SWGG and VA algorithms are respectively shown in Figure 1(a), Figure 1(a), Figure 2(c) and Figure 1(d). In Figure 9, for instance, we compare two cases: *BlockSize*=2×2 vs *BlockSize*=*DataSize*. For the regular algorithms SWLAG and SA, since each computing cell only depends on its three adjacent cells, there is little difference in the number of cache misses between sweeping the elements completely row by row when *BlockSize*=*DataSize* and computing block by block when *BlockSize*=2×2. For the irregular algorithm SWGG, since each computing cell depends on all of its upper and left cells, the number of cache misses is quite different for varied block sizes. We take block 2 (labeled in Figure 9) for instance. That computing row by row when *BlockSize*=*DataSize* may make all the upper cell data needed for computing the cell $a_1$ be removed away from the cache when computing cell $a_3$, causing a cache miss in such case. While that computing block by block when *BlockSize*=2×2 could reuse all the upper cell data needed for computing the cell $a_1$ when computing cell $a_3$, thus no cache miss occurs in such case. Therefore, the smaller suitable block size can be used to reduce the number of cache misses in some cases, just as shown in Figure 8(c,g). For the irregular algorithm VA, since each cell depends on all those completed cell, those data needed for computing cell $a_1$ could be reused in the computation of $a_3$ both when *BlockSize*=*DataSize* and when *BlockSize*=2×2. It means there are few differences as to the number of cache misses between sweeping the elements completely row by row when *BlockSize*=*DataSize* and computing elements block by block when *BlockSize*=2×2 for the algorithm VA.

However, Figures 8(e,f,h) illustrate that the value of cache miss rate decreases as the value of argument *BlockSize* increases. The EasyPDP overhead is a critical issue here. Specifically, the smaller division of data block would incur a larger DAG size needed in EasyPDP(more overhead), which in turn brings about more number of cache misses when operating the DAG. Therefore, the big value of argument *BlockSize* is a key way in reducing the number of cache misses incurred by its overhead.

In conclusion, for the algorithms SWLAG, SA, VA, the block size has a few contribution to the number of cache misses from the feature of the algorithm itself, while the overhead incurred by the argument *BlockSize* is a key contributor in affecting the cache miss of EasyPDP. Thus, the cache miss rate decreases as the value of argument *BlockSize* increases for these algorithms. In contrast, the character of SWGG algorithm is a dominate factor which dramatically affects the number of cache misses in comparison to its overhead. Therefore, the smaller block size reduces the cache miss rate for the

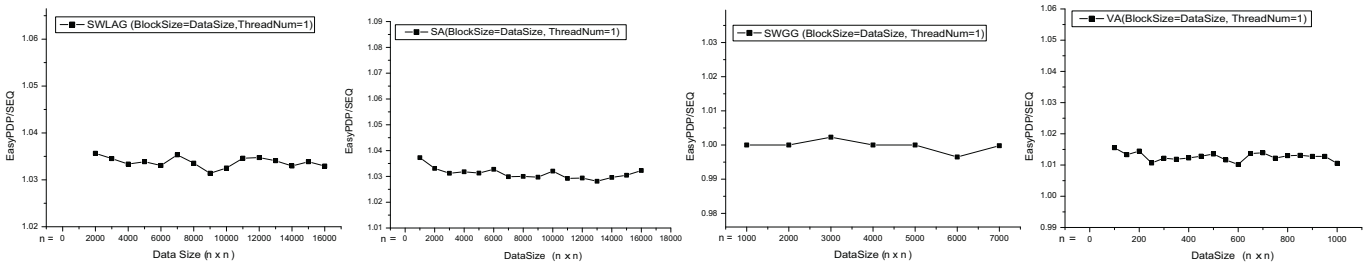



Fig. 7: The EasyPDP overhead for regular and irregular DP algorithms. By setting *ThreadNum*=1 and *BlockSize=DataSize*, we compare the run time of EasyPDP to that of the sequential iterative code with various input data sizes for four DP algorithms, aiming to see the overhead of EasyPDP system in the multi-core environment.

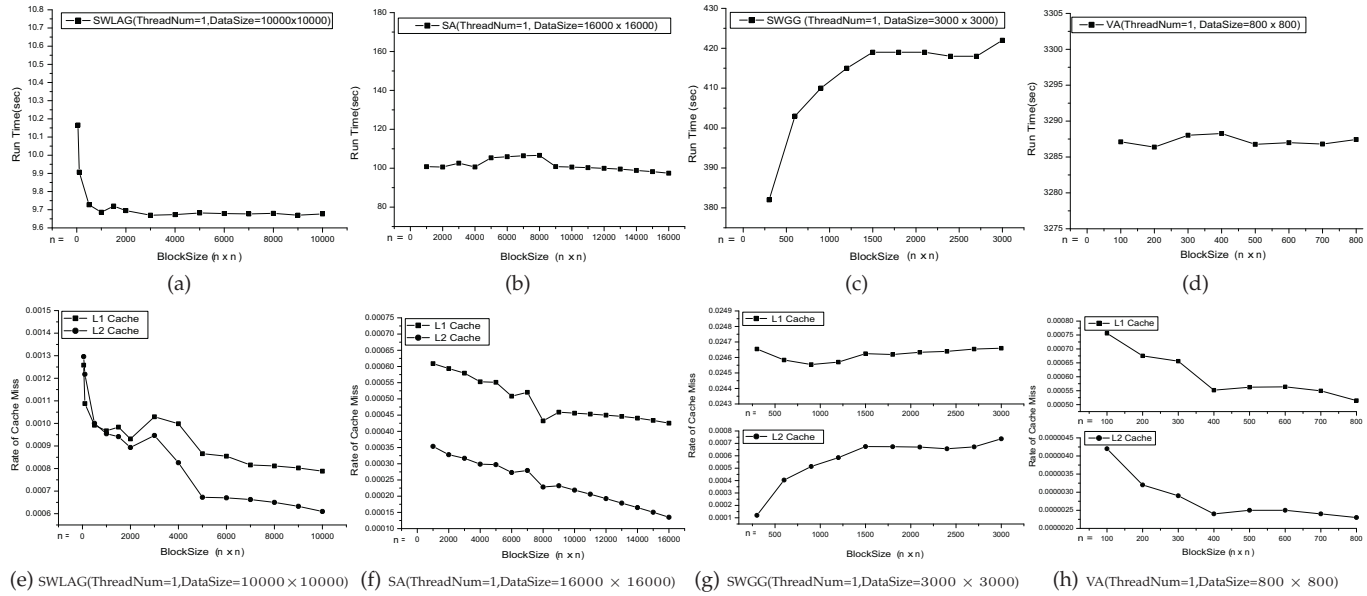


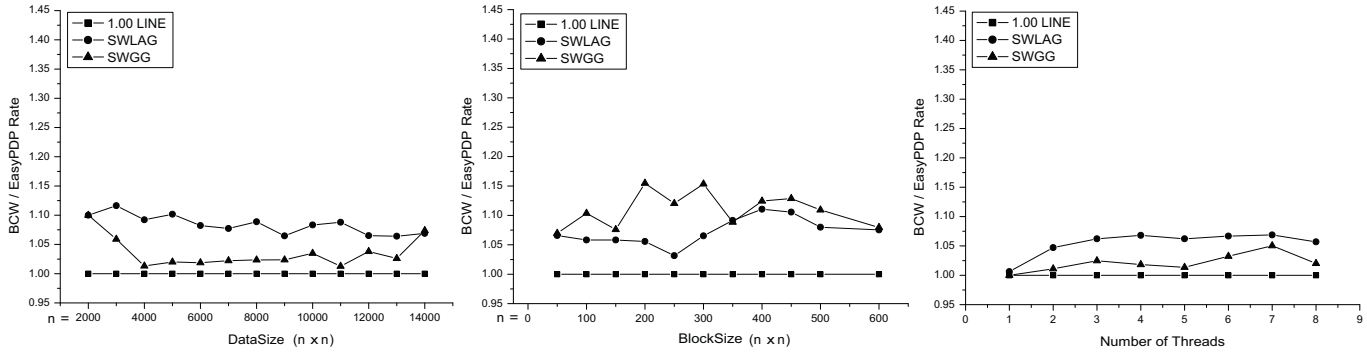

Fig. 8: The EasyPDP cache miss for regular and irregular DP algorithms. Figures from (a) to (d), through the changes of BlockSize when ThreadNum=1, are aimed to see the affection of cache miss in EasyPDP for regular and irregular DP algorithms. Figures from (e) to (h) are the rate of cache miss results sampled with profiler vtune for L1 and L2 caches.

algorithm SWGG.

### C.5 Comparison to BCW

In contrast with dynamic runtime system EasyPDP, the Column based Wavefront(CW)[10] and Block-Cycle based Wavefront(BCW)[9] are static data partitioning methods to parallelize DP algorithm. The CW algorithm can be viewed as a special BCW algorithm when setting the argument *block_col* to the result of *data_col* divided by *thread_num* for BCW. Thereby it only needs to compare the performance with BCW for EasyPDP. To the BCW algorithm implementation, we also adopt the DAG Data Driven Model, but the static worker pool is considered here. In order to compare the EasyPDP with BCW thoroughly and completely, we define the metric *BCW/EasyPDP rate* as BCW divided by EasyPDP with their run times in the same condition, and do comparisons from three perspectives: data size, block size and the number of threads respectively.

Figure 10 presents the comparison between the rate results of EasyPDP and BCW. The baseline *1.00 LINE* is given. It indicates EasyPDP is better if the rate points are above the baseline, otherwise it is BCW. A conclusion drawn from the diagram is that EasyPDP is more efficient than BCW both for regular and irregular DP algorithms, as it can be observed that the experimental rate curves for both regular and irregular DP algorithms are all above the *1.00 LINE*. The primary reason is that, for the DP algorithms, its data dependency is extremely strong, which means that only a few DAG block nodes are computable at the beginning, and more and more new computable nodes are spawn during the running process. Compared with EasyPDP, the BCW, which uses the static data allocation and scheduling




(a) (b) (c)

Fig. 10: The compared BCW/EasyPDP rate results from three different aspects: data size, block size and the number of threads, as shown in Figures (a),(b),(c) respectively. The EasyPDP is more efficient than BCW when the point is above the baseline *1.00 LINE*, otherwise the BCW is more efficient.

method, has a fatal situation case during the runtime that there are some computable DAG nodes as well as some idle threads simultaneously, whereas the case will never occur for EasyPDP, which uses a dynamic data allocation and scheduling runtime system. Moreover, for the BCW, the case often occurs many times and even more, especially for irregular DP algorithms.

### C.6 Fault Recovery

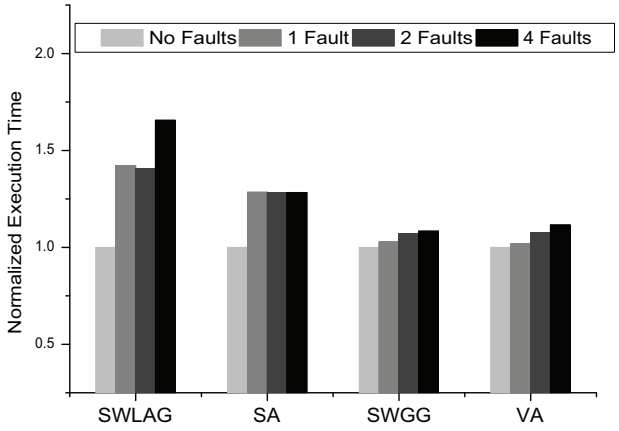



Fig. 11: Normalized execution time in the presence of different number of faults for EasyPDP.

Figure 11 presents results for fault injection experiments for EasyPDP. The graphs represent normalized execution time. An error may occur at an arbitrary point within the program execution, which may affect the execution and buffers, but does not corrupt the runtime or its data structures. The EasyPDP runtime detects faults through timeouts and recovers to complete the execution correctly. The whole process of fault tolerance and recovery is completely transparent to the programmer. The initial value for the argument *timeout* is set by the user and dynamically changed at runtime. Its value can greatly affect the system performance (See discussion in Section 4.2.4). Here the *timeout* value is initialized to 10 seconds. Note that fault impacts for the regular DP algorithms (SWLAG and SA) are more prominent than the irregular DP algorithms(SWGG and VA), due to the fact that the execution times for the irregular DP algorithms are far greater than the regular ones for the same data size. Here the *timeout* value of 10 is suitable for the irregular DP algorithms, while it is relatively too large compared with the execution times of the regular DP algorithms and hence impacts the performance greatly.

## APPENDIX D
## PROOF OF THEOREM 1

**Theorem 1.** *When $t = t_0(t_0 \geq 1)$, the optimal value of $b$ for the minimal value $S$ is*

$$b = \begin{cases} d & (t_0 = 1) \\ \sqrt{\frac{c_0 \times d}{k \times t_0 \times (t_0 - 1)}} & (t_0 > 1) \end{cases} \quad (1)$$

*Therefore, the optimal minimal value $S$ is*

$$S = \begin{cases} d \times k + c_0 & (t_0 = 1) \\ \frac{\sqrt{t_0 \times (t_0-1) \times c_0 \times d \times k} + d \times k \times \sqrt{t_0}}{t_0} + & \\ c_0 \times (t_0 - 1) + & \\ \sqrt{c_0 \times d \times k \times (t_0 - 1)} & (t_0 > 1) \end{cases} \quad (2)$$

**Proof**. When $t = t_0(t_0 \geq 1)$, the value of $S$ is

$$S = (b \times k + c_0) \times (t_0 - 1 + \frac{d}{b \times t_0}) \quad (3)$$

$$\therefore \frac{\partial S}{\partial b} = k \times (t_0 - 1 + \frac{d}{b \times t_0}) + (b \times k + c_0) \times (-\frac{d}{b^2 \times t_0}) = k \times (t_0 - 1) - \frac{c_0 \times d}{b^2 \times t_0} \quad (4)$$

**(i)**. In the case when $t_0 = 1$, we have

$$\frac{\partial S}{\partial b} = -\frac{c_0 \times d}{b^2} < 0 \quad (5)$$

Hence, it means that $\frac{\partial S}{\partial b}$ is a monotonic decreasing function, thereby we have the optimal minimal value $S$ when $b = d$. In this case, the optimal minimal value $S$ is:

$$S = d \times k + c_0 \tag{6}$$

**(ii)**. In the case when $t_0 > 1$, we can make $\frac{\partial S}{\partial b} = 0$, then it has

$$b = \sqrt{\frac{c_0 \times d}{k \times t_0 \times (t_0 - 1)}} \tag{7}$$

In this case, the optimal minimal value $S$ is:

$$S = \frac{\sqrt{t_0 \times (t_0 - 1) \times c_0 \times d \times k} + d \times k \times \sqrt{t_0}}{t_0} + c_0 \times (t_0 - 1) + \sqrt{c_0 \times d \times k \times (t_0 - 1)}. \tag{8}$$

**(iii)**. In terms of **(i)** and **(ii)**, we therefore obtain

$$b = \begin{cases} d & (t_0 = 1) \\ \sqrt{\frac{c_0 \times d}{k \times t_0 \times (t_0 - 1)}} & (t_0 > 1) \end{cases}$$

$$S = \begin{cases} d \times k + c_0 & (t_0 = 1) \\ \frac{\sqrt{t_0 \times (t_0 - 1) \times c_0 \times d \times k} + d \times k \times \sqrt{t_0}}{t_0} + c_0 \times (t_0 - 1) + \sqrt{c_0 \times d \times k \times (t_0 - 1)} & (t_0 > 1) \end{cases}$$

## APPENDIX E
## PROOF OF THEOREM 2

**Theorem 2.** *When $b = b_0 (0 < b_0 \leq d)$, the optimal value of $t$ for the minimal value $S$ is*

$$t = \sqrt{\frac{d}{b_0}} \quad (0 < b_0 \leq d) \tag{9}$$

*in this case, the optimal minimal value $S$ is*

$$S = (b_0 \times k + c_0) \times (2 \times \sqrt{\frac{d}{b_0}} - 1) \quad (0 < b_0 \leq d) \tag{10}$$

**Proof**. When $b = b_0 (0 < b_0 \leq d)$, the value of $S$ is

$$S = (b_0 \times k + c_0) \times (t - 1 + \frac{d}{b_0 \times t}) \tag{11}$$

$$\therefore \frac{\partial S}{\partial t} = (b_0 \times k + c_0) \times (1 - \frac{d}{b_0 \times t^2}) \tag{12}$$

Let $\frac{\partial S}{\partial t} = 0$, we can get

$$t = \sqrt{\frac{d}{b_0}} \tag{13}$$

In this case, the optimal minimal value $S$ is

$$S = (b_0 \times k + c_0) \times (2 \times \sqrt{\frac{d}{b_0}} - 1) \qquad (0 < b_0 \leq d) \tag{14}$$

## APPENDIX F
## PROOF OF THEOREM 3

**Theorem 3.** *The optimal value of timeout $r$ for the DP algorithm is*

$$r = \begin{cases} (\lceil \frac{\sqrt{\frac{d}{b}} - t}{t} \rceil + 1) \times (b \times k + c_0) + \varepsilon & (t < \sqrt{\frac{d}{b}}, 0 < b \leq d) \\ (b \times k + c_0) + \varepsilon & (t \geq \sqrt{\frac{d}{b}}, 0 < b \leq d) \end{cases} \tag{15}$$

*where $\varepsilon (\varepsilon > 0)$ represents a small extra necessary delayed time for the fault tolerance and recovery mechanism.*

**Proof**. In the EasyPDP, the cost of a DAG node task consists of two parts: computation cost ($C_{ij}$ for short) and waiting cost ($W_{ij}$ for short) in pool queue.

**(i)**. *The optimal value of timeout should equal to the maximum cost of all DAG node tasks by adding a small extra necessary delayed time ( $\varepsilon$ for short, $(\varepsilon > 0)$) for the fault tolerance and recovery mechanism. That is,*

$$r = \max_{0 \leq i,j < \sqrt{\frac{d}{b}}} \{C_{ij} + W_{ij} + \varepsilon\}. \tag{16}$$

Next, we first prove the optimal value of

$$r = \max_{0 \leq i,j < \sqrt{\frac{d}{b}}} \{C_{ij} + W_{ij}\} \tag{17}$$

from the ideal theoretic aspect without considering the necessary delayed time for the fault tolerance and recovery mechanism.

**(1)**. $r \not< \max_{0 \leq i,j < \sqrt{\frac{d}{b}}} \{C_{ij} + W_{ij}\}$. (18)
*Prove*: If $r$ is less than the maximum cost of all DAG node tasks, the fault tolerance and recovery mechanism will then wrongly kill some being computed node tasks such as the node task of the maximum cost, makes that task recomputed and affects the system performance. So $r$ can't be less than the maximum cost of all DAG node tasks.

**(2)**. *It will make the fault tolerance and recovery mechanism obtuse when* $r > \max_{0 \leq i,j < \sqrt{\frac{d}{b}}} \{C_{ij} + W_{ij}\}$. (19)
*Prove*: If $r$ is larger than the maximum cost of all DAG node tasks, it means that the fault tolerance and recovery mechanism will obtusely waste at least $(r - \max_{0 \leq i,j < \sqrt{\frac{d}{b}}} \{C_{ij} + W_{ij}\})$ time to detect faults when a fault occurred in a random DAG node task. In this case, the system performance will be affected, too.

**(3)**. In terms of **(1)** and **(2)** above, we could obtain from the ideal theoretic aspect that

$$r = \max_{0 \leq i,j < \sqrt{\frac{d}{b}}} \{C_{ij} + W_{ij}\}.$$

However, we should consider a small extra necessary delayed time for the fault tolerance and recovery mechanism from the implementation aspect in order to guarantee correct fault detection.

$$\therefore r = \max_{0 \le i,j < \sqrt{\frac{d}{b}}} \{C_{ij} + W_{ij} + \varepsilon\}.$$

**(ii)**. *When* $t < \sqrt{\frac{d}{b}}$ *, the optimal value of* $r = (\lceil \frac{\sqrt{\frac{d}{b}} - t}{t} \rceil + 1) \times (b \times k + c_0) + \varepsilon$. (20)

**(1)**. *For the case when* $t < \sqrt{\frac{d}{b}}$ *, there are at most* $\sqrt{\frac{d}{b}}$ *computable DAG nodes at any time.*

*Prove*: Let's consider the dependence relationship among DAG nodes. There is at most one computable DAG node in a row or column of the DAG at any time. As each DAG node depends on its upper and left adjacent nodes, it will violate the dependence relationship if there are more than one computable DAG nodes in a row or column of DAG at the same time. Therefore, the maximum number of computable DAG nodes at any time should equal to the number of rows or columns of DAG. That is, there are at most $\sqrt{\frac{d}{b}}$ computable DAG nodes at any time when $t < \sqrt{\frac{d}{b}}$.

**(2)**. Consider the case when there are $\sqrt{\frac{d}{b}}$ computable node tasks, where the average computation cost for each node task is $C = (b \times k + c_0)$. Since there are $t$ threads, therefore there are $t$ tasks being completed and $(\sqrt{\frac{d}{b}} - t)$ tasks remained in the pool queue. For that remained node tasks, it needs to wait at most $\lceil \frac{\sqrt{\frac{d}{b}} - t}{t} \rceil$ number of DAG node computing time in the pool queue before being computed if we assume that each time there are $t$ tasks completed. Let's assume $f : f(i,j) = 1$ when node $(i,j)$ is the remained node task; otherwise, $f(i,j) = 0$. Moreover, for the remained nodes, $g(i,j)$ denotes the number of DAG nodes including node $(i,j)$ itself queued in the buffer when node $(i,j)$ is distributed to the pool queue. Therefore,

$$W_{ij} = \begin{cases} 0 & f(i,j) = 0 \\ \lceil \frac{g(i,j)}{t} \rceil \times C & f(i,j) = 1 \end{cases}, \quad (21)$$

$$(0 \le i,j < \sqrt{\frac{d}{b}}, 1 \le g(i,j) \le \sqrt{\frac{d}{b}} - t)$$

$$\therefore \max_{0 \le i,j < \sqrt{\frac{d}{b}}} \{C_{ij} + W_{ij}\} = \lceil \frac{\sqrt{\frac{d}{b}} - t}{t} \rceil \times C + C$$

$$= (\lceil \frac{\sqrt{\frac{d}{b}} - t}{t} \rceil + 1) \times (b \times k + c_0).$$

$$\therefore r = \max_{0 \le i,j < \sqrt{\frac{d}{b}}} \{C_{ij} + W_{ij} + \varepsilon\}$$

$$= (\lceil \frac{\sqrt{\frac{d}{b}} - t}{t} \rceil + 1) \times (b \times k + c_0) + \varepsilon.$$

**(iii)**. *When* $t \ge \sqrt{\frac{d}{b}}$ *, the optimal value of* $r = (b \times k + c_0) + \varepsilon$. (22)

*Prove*: When $t \ge \sqrt{\frac{d}{b}}$ and average computation cost for each node task is $C = (b \times k + c_0)$, there are at most $t$ computable DAG nodes along the anti-diagonal at any time. It implies that no node tasks need to be waited in the pool queue before other node tasks completed in this case. Therefore, we have $f(i,j) = 0, W_{ij} = 0 (0 \le i,j < \sqrt{\frac{d}{b}})$ for all nodes.

$$\therefore \max_{0 \le i,j < \sqrt{\frac{d}{b}}} \{C_{ij} + W_{ij}\} = C + 0 = (b \times k + c_0). \quad (23)$$

$$\therefore r = \max_{0 \le i,j < \sqrt{\frac{d}{b}}} \{C_{ij} + W_{ij} + \varepsilon\} = (b \times k + c_0) + \varepsilon. \quad (24)$$

**(iiii)**. In terms of **(ii)** and **(iii)**, we could obtain

$$r = \begin{cases} (\lceil \frac{\sqrt{\frac{d}{b}} - t}{t} \rceil + 1) \times (b \times k + c_0) + \varepsilon & (t < \sqrt{\frac{d}{b}}) \\ (b \times k + c_0) + \varepsilon & (t \ge \sqrt{\frac{d}{b}}) \end{cases}$$

## REFERENCES


[1] J. Bowie, R. Luthy, and D. Eisenberg, "A Method to Identify Protein Sequences That Fold Into A Known Three-dimensional Structure," *Science*, vol. 253, no. 5016, pp. 164-170, 1991.

[2] C. Ranger, R. Raghuraman, A. Penmetsa, et al., "Evaluating MapReduce for Multi-core and Multiprocessor Systems," *International Symposium on High-Performance Computer Architecture*, pp. 13-24, 2007.

[3] E. Kohler, R. Morris, and B. Chen, "Programming Language Optimizations for Modular Router Configurations," *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pp. 251-263, 2002.

[4] B.D. Carlstrom, A. McDonald, H. Chafi, et al., "The ATOMO Transactional Programming Language," *2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1-13, June 2006.

[5] T. Harris and K. Fraser, "Language Support for Lightweight Transactions," *Proceedings of the 2003 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

[6] S. Balakrishnan and G.S. Sohi, "Program Demultiplexing: Data-flow based Speculative Parallelization of Methods in Sequential Programs," *33rd International Symposium on Computer Architecture, ISCA 2006*, June 2006.

[7] C. Ciressan, E. Sanchez, M. Rajman, and J.C. Chappelier, "An FPGA-based coprocessor for the parsing of context-free grammars," *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[8] M. Farach and M. Thorup, "Optimal evolutionary tree comparison by sparse dynamic programming," *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pp. 770-779, 1994.

[9] W.G Liu and B. Schmidt, "Parallel design pattern for computational biology and scientific computing applications," *Proceedings. IEEE International Conference on Cluster Computing*, pp. 456-459, 2003.

[10] V. Kumar and A. Grama, A. Gupa, and G. Karypis, *Introduction to Parallel Computing*, The Benjamin-Cummings Publishing Company, Inc., 1994.

[11] R.A. Chowdhury and V. Ramachandran, "Cache-efficient dynamic programming algorithms for multicores," *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*, pp. 207-216, 2008.

[12] R.A. Chowdhury, H.S. Le, and V. Ramachandran, "Cache-oblivious dynamic programming for bioinformatics," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 7, no. 3, pp. 495-510, 2009.

[13] R.A. Chowdhury and V. Ramachandran, "Cache-oblivious dynamic programming," *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 591-600, 2006.

[14] R.A. Chowdhury, H. Le, and V. Ramachandran, *Efficient cache-oblivious string algorithms for Bioinformatics*, Technical Report TR-07-03, Dept. of Computer Sciences, University of Texas at Austin, February 2007.

[15] G. Blelloch, R. Chowdhury, P. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch, "Provably good multicore cache performance for divide-and-conquer algorithms," *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 501-510, 2008.

[16] Z. Galil and K. Park, "Dynamic Programming with Convexity, Concavity and Sparsity," *Theoretical Computer Science*, vol. 92, pp. 49-76, 1992.

[17] X. Huang and K.M. Chao, "A Generalized Global Alignment Algorithm," *Bioinformatics*, vol. 19, no. 2, pp. 228-233, 2003.

[18] N. Futamura, S. Aluru, X. Huang, "Parallel Syntenic Alignments," *HiPC '02 Proceedings of the 9th International Conference on High Performance Computing*, pp. 420-430, 2002.

[19] T. Smith and M. Waterman, "Identification of Common Molecular Subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195-197, 1981.

[20] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Protein and Nucleic Acids*, Cambridge Univ. Press, 1998.

[21] R. Nussinov, G. Pieczenik, J. R. Griggs, D. J Kleitman, "Algorithms for loop matchings," *SIAM Journal on Applied Mathematics*, vol. 35, no. 1, pp. 68-82, 1978.

[22] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260-269, 1967.

[23] D.W. Mount, *Bioinformatics-Sequence and Genome Analysis*, Cold Spring Harbor Laboratory Press, 2001.

[24] M.S. Gelfand, A.A. Mironov, and P.A. Pevzner, "Gene Recognition Via Spliced Sequence Alignment," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 93, no. 17, pp. 9061-9066, 1996.

[25] M. Zuker and P. Stiegler, "Optimal Computer Folding of Large RNA Sequences Using Thermodynamics and Auxiliary Information," *Nucleic Acids Research*, vol. 9, no. 1, pp. 133-148, 1981.

[26] B.Lewis and D.J. Berg. *Multithreaded Programming with Pthreads*, Prentice Hall, 1998.

[27] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K.S. Kim, "Robust System Design with Built-In Soft-Error Resilience," *Computer*, vol 38, no. 2, pp. 43-52, 2005.

[28] J. C. Smolens et al., "Fingerprinting: Bounding Soft-error Detection Latency and Bandwidth," *11th International Conference on Architectural Support for Programming, Languages and Operating Systems*, pp. 224-234, Oct. 2004.

[29] P. Edmonds, E. Chu, and A. George, "Dynamic Programming on a Shared Memory Multiprocessor," *Parallel Computing*, vol. 19, no. 1, pp. 9-22, 1993.

[30] Z. Galil and K. Park, "Parallel Algorithm for Dynamic Programming Recurrences with More Than O(1) Dependency," *Journal of Parallel and Distributed Computing*, vol. 21, no. 2, pp. 213-222, 1994.

[31] P.G. Bradford, "Efficient Parallel Dynamic Programming," *Proc. 30th Ann. Allerton Conf. Comm. Control and Computing*, pp. 185-194, 1992.

[32] A. Mark and S. Ramesh, "PC software performance tuning," *Computer*, vol. 29, no. 8, pp. 47-54, 1996.

[33] G.M. Tan, H.N. Sun and R.G. Gao, "A parallel dynamic programming algorithm on a multi-core architecture," *Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 135-144, 2007.

[34] G.M. Tan et al., "Locality and Parallelism Optimization for Dynamic Programming Algorithm in Bioinformatics," *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC'06*, pp. 11-17, 2006.

[35] F. Almeida, R. Andonov, and D. Gonzalez, "Optimal Tiling for RNA Base Pairing Problem," *Proc. 14th Ann. ACM Symp. Parallel Architecture and Algorithm (SPAA 02)*, pp. 173-182, 2002.

[36] W. Zhou and D.K. Lowenthal, "A Parallel, Out-of-Core Algorithm for RNA Secondary Structure Prediction," *Proc. 35th Intl Conf. Parallel Processing (ICPP 06)*, pp. 74-81, 2006.

[37] J.S. Vitter, "External Memory Algorithms and Data Structures: Dealing with Massive Data," *ACM Computing Surveys*, vol. 33, no. 2, pp. 209-271, 2001.

[38] W. Liu and B. Schmidt, "A Generic Parallel Pattern-Based System for Bioinformatics," *Proc. EURO-PAR*, pp. 989-996,2004.

[39] W. Liu and B. Schmidt, "Parallel Pattern-Based Systems for Computational Biology: A Case Study," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 8, pp. 750-763, 2006.

[40] J. Dean and J. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no.1 , pp. 107-113, 2008.

[41] M. I. Gordon et al., "A Stream Compiler for Communication-exposed Architectures," *In the Proceedings of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 291-303, Oct. 2002.

[42] D. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, vol. 18, no. 6. pp. 341-343, 1975.

[43] X. Huang, "A space-efficient parallel sequence comparison algorithm for a message-passing multiprocessor," *International Journal of Parallel Programming*, vol. 18, no. 3, pp. 223-239, 1989.

[44] S. Rajko, S. Aluru, "Space and time optimal parallel sequence alignments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 12, pp. 1070-1081, 2004.

[45] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," *Theory of Computing Systems*, vol. 35, no. 3, pp. 321-347, 2002.

[46] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," *ACM SIGPLAN Notices*, pp. 212-223, 1998.

[47] G. E. Blelloch and P. B. Gibbons, "Effectively sharing a cache among threads," *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pp. 235-244, 2004.

[48] G. E. Blelloch, P. B. Gibbons, and Y. Matias, "Provably efficient scheduling for languages with fine-grained parallelism," *Journal of the ACM*, vol. 46, no. 2, pp. 281-321, 1999.

[49] G. E. Blelloch, P. B. Gibbons, G. J. Narlikar, and Y. Matias, "Space-efficient scheduling of parallelism with synchronization variables," *Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 12-23, 1997.